Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 49 (2015) 211 – 219

Procedia Computer Science

ICAC3'15

Improved Parallel Lexical Analysis Using OpenMP on Multi-Core Machines

Amit Barve[a], Brijendra Kumar Joshi[b]

[a]*Assitant Professor, CSE Dept. VIIT, Pune India,*
[b]*Professor,MCTE, Mhow, India*

Abstract

Lexical Analysis is the first and foremost step of a compiler. Various attempts have been made to improve the lexical analysis phase by exploiting the inherent parallel processing capability of multi-core machines. In this paper we present an approach for doing parallel lexical analysis using OpenMP. We demonstrate the improvement in lexical analysis phase by automatically generating C programs having up to 10,000 potentially parallel constructs like if..else, for, while loops, switch..case etc. The maximum speedup achieved for 7 CPUs is 6.84.

Peer-review under responsibility of organizing committee of the 4th International Conference on Advances in Computing, Communication and Control (ICAC3'15)

Keywords: Parallel Lexical Analysis; Multi-Core Machines; OpenMP.

1. Introduction

In present times, compilers and high level languages are the key stones and also the establishment of intricate and universal programming foundation that drives the worldwide economy. As per Hall et al [1], in the present decade i.e. 2010 to 2020 compiler research will play a critical role in addressing two of the major challenges facing the overall computer field:

(a) Security and reliability of complex software systems.
(b) Cost of programming multi-core processors.

Peer-review under responsibility of organizing committee of the 4th International Conference on Advances in Computing, Communication and Control (ICAC3'15)
doi:10.1016/j.procs.2015.04.246

Regardless of the way that the clock rate of the machine has very nearly arrived at its hypothetical cutoff, the machine velocity is as of now keeping on increasing astonishingly because of expanded parallelism as multi-core execution units. This pattern of parallel structural engineering is one of the best difficulties for the processor and programming commercial ventures.

To meet the challenges posed by the multi-core machines the compiler field is also thriving to reap benefits of wide spread use of high level languages. In doing so, it is essential that we make parallel programming mainstream. To improve the productivity, it is essential that parallel languages and their compilers be developed to fully exploit the new features of modern processors specifically multi-cores.

A Compiler is a complex program that translates a code written in one language (Source Language) to another language (Target Language). A compiler has always followed a sequential pattern mainly consisting of phases of lexical analysis, syntax analysis, intermediate code generation, code optimization and code generation. All these phases have been well described in popular texts on the subject like Gries[2], Tremblay[3], Holub[4], Aho et al[5], Aho[6], etc.

2. Lexical Analysis Phase

The main task of the lexical analysis phase is to read the input characters of the source program, group them into meaningful sequences called lexemes, and produce as output a token for each lexeme in the source program. The process of splitting input into tokens is called tokenizing or scanning.

The stream of tokens is sent to the parser for syntax analysis. It is common for the lexical analyzer to interact with the symbol table as well. When the lexical analyzer discovers a lexeme constituting an identifier, it needs to enter that lexeme into the symbol table. In some cases, information regarding the kind of identifier may be read from the symbol table by the lexical analyzer to assist it in determining the proper token it must pass to the parser.

For the first time Lesk and Schmidt took the challenge of generating lexical analyzers from specifications in the form of regular expressions. Their tool LEX [7] is by far the most important in compiler development. Currently Fast Lexical Analyzer Generator (FLEX) [8] is normally used for automatic generation of lexical analyzers.

3. Parallel Lexical Analysis

People have suggested to parallelize the lexical analysis phase by splitting the source code on some criteria or the other. Daniele and Gregory[9] compiled the flex kernel and ran it on each of the Synergetic Processing Element, on IBM cell processor. Umarani developed a parallel lexical analyzer for cell processor [10]. His approach was to split the source code into fixed size blocks using dynamic block splitting algorithm. Mickunas and Shell[11] proposed to split lexical analysis into scanning and screening. Barve & Joshi presented three algorithms to enhance the lexical analysis phase based on splitting source code on the basis of potentially parallel constructs [12], source code lines and white space characters [13].

4. Parallelizing A Program Using Openmp

OpenMP is a programming standard designed to support multi-platform shared-memory parallel programming in C/C++ and Fortran. It defines number of APIs for a portable, scalable model for developing parallel applications. Parallelism can be achieved by writing the following code in a C program:

```
#pragma omp parallel ….
```

By introducing the above statement in a C program, the program is now capable to execute in parallel by identifying the total number of threads present in the machine. By default, OpenMP uses all the threads for execution but by choosing the option user can limit the thread usage. Using OpenMP is good option to parallelize the program execution but it is the responsibility of programmers to identify the locations in the program which can

be parallelized. For example, consider the following pseudo-code:

```
for (int x=0; x < width; x++)
{
    for (int y=0; y < height; y++)
      {
          finalImage[x][y] = RenderPixel(x,y, &sceneData);
      }
}
```

Fig. 1: Pseudo code

This piece of code simply goes through each pixel of the screen, and calls a function, RenderPixel, to determine the final color of that pixel. Note that the results are simply stored in an array. Simply put, the entire scene that is being rendered is stored in a variable, sceneData, whose address is passed to the RenderPixel function. Because each pixel is independent of all other pixels, and because RenderPixel is expected to take a noticeable amount of time, this small snippet of code is a prime candidate for parallelization with OpenMP. Consider the following modified pseudo-code:

```
#pragma omp parallel for
for (int x=0; x < width; x++)
{
    for(int y=0; y < height; y++)
        {
            finalImage[x][y]=RenderPixel(x,y,&sceneData);
        }
}
```

Fig. 2: Modified pseudo code with OpenMP

The only change to the code is the line directly above the outer for loop. This compiler directive tells the compiler to auto-parallelize the for loop with OpenMP. If a user is using a quad-core processor, the performance of a code fragment can be expected to be many folds with the inclusion of just one line of code. In practice, true linear or super linear speedups are seldom, whereas near linear speedups are common. The details of OpenMP can be found in [14].

5. Speed-Up

The parallel methodologies in programming are designed with an aim that parallel programs execute faster as compared to sequential ones. Speed up is the ratio of sequential and parallel execution times. It can be represented as:

$$Speedup = \frac{\text{Sequential exectution time}}{\text{Parallel exection time}} \quad (1)$$

The operations performed by a parallel algorithm can be put into three categories [15]:

(a) Operations that must be performed sequentially.
(b) Operations that can be performed in parallel.
(c) Operations requiring communication among processors.

Amdhal's Law:

According to Amdahl's law speed up φ is defined as:

$$\varphi \leq \frac{1}{f+(1-f)/p} \quad (2)$$

Where, f is the fraction of operations in a computation that must be performed sequentially, and $0<=f<=1$. P is the number of processors available.
For example consider the following *C* code:

```
1. void main()
2. {
3. int a, b, c, d, e;
4. c = a + b;
5. d = a - b;
6. e = c / d;
7. }
```

Fig. 3: C code

In this *C* code line numbers 1-3 are standard *C* code lines. Line number 4-5 can be executed in parallel because a & b are independent of each other, line number 6 cannot be executed in parallel because it is dependent on results of c and d. Line number 7 also executes sequentially.
For the given program f can be calculated as given below:

- Total number of lines = 7
- Number of lines that must be executed sequentially = line numbers 1, 2, 3, 6, 7 =5
- Fraction of lines that must be executed sequentially f = 5 / 7
- Number of processor = 7
- So, speedup for 7 processors will be:

$$\text{Speedup} = \frac{1}{5/7+(1-5/7)/7} = 1.33 \quad (3)$$

In our case, the Amdhal's law for calculating speed up is not applicable because Amdhal's law talks about *execution* in parallel. We are performing lexical analysis in parallel. For this reason if we consider the same example, lexical analysis of all seven lines can be done in parallel by assigning each line to an individual processor as there is no inter dependency. But there can be some problems in parallel lexical analysis done in this way. Consider the source code of Fig. 3 re-written as in Fig. 4.

```
1.  void main
2.  ()
3.  {
4.  int a, b, c, d, e;
5.  c = a + b;
6.  d = a - b;
7.  e = c / d;
8.  }
```

Figure 4: C code with modifications

The line numbers 1 and 2 are assigned to different processors for lexical analysis then for subsequent phase arranging tokens in proper sequence would be a daunting tasking. Therefore we take care of such situations in our algorithm.

Let us call potentially parallel constructs like for, switch..case, while etc. PPC for simplicity. We assign all lines of code that are outside PPCs to a single CPU and all PPCs to individual CPUs. So, we consider all outside PPC code to be analyzed in parallel with PPCs. For example, if we have 6 PPCs and 7 CPUs with us then one CPU would analyze non-PPC code and remaining 6 CPUs would analyze one PPC each in parallel. So, all 7 CPUs do lexical analysis in parallel. In view of the foregoing, we defined the speedup as:

$$Speedup = \frac{\text{Time taken in complete lexical analysis by one CPU}}{\text{Max(t1, t2, … … . . … tn)}} \quad (4)$$

Where, ti is the time taken by i^{th} CPU for lexical analysis of code block assigned to it.

6. Parallel Lexical Analysis Using Memory Blocks

Now we present the improved version of the algorithm for parallel lexical analysis developed by the authors earlier. The algorithms developed by them write source code block markers into a text file which is read later and based on these markers number of processes are forked and assigned to different CPUs[12][13]. The assignment of processors is done using processor affinity concept[16][17]. The earlier algorithm used round robin technique to assign processes to CPUs. Since there is only single read head available on the disk the waiting time in I/O queue increases substantially for every processor.

We can improve the performance of the algorithm by creating the blocks in memory and processing each block in parallel. This avoids waiting time for a process in I/O queue. To achieve the forgoing, memory blocks of the construct size is created and processed in parallel using OpenMP. The Improved version of the algorithm is shown in Fig. 5.

1. Read Source file say *source.c* and detect Constructs. While doing so gather the following information in a text file say *source.txt***:** (a) Starting Byte of Constructs, say,*start***.** (b) Ending Byte of Constructs, say,*end***.**
2. Open *source.c* in read only mode.
3. For each line written in source.txt prepare block of size *end-start*+1 into memory. Copy the contents of construct into memory blocks.
4. Call *yylex()* on each memory block in parallel.

Figure 5. Parallel Lexical Analysis using Memory Block

The code for proposed algorithm is given in appendix ‘A’.

7. Experimental Results

The experiment based on the above algorithm was carried out on Ubuntu 12.04 LTS on Wipro Netpower server with Intel Xeon E5606 base duel CPU Quad Core machine with 8MB on chip cache and 24 GB RAM and processor speed 2.13 GHz having 8 cores in total. For testing the algorithms, source files of different sizes and different number of loops and decision constructs were required. For this, a C program was written [18] which generates random number of variables of type *int.* Variable names generated were t1, t2,…, t40. Random number (from interval [1, 100]) of loops and decision constructs were generated using simple assignment statements and simple *printf()* generated randomly. Number of iterations of each loop was also automatically fixed with a random number from interval [1, 20]. Size of each loop and decision constructs (i.e. number of assignment statements) so generated were also random from the interval [1, 20]. Source programs were generated with 10 to 10000 constructs. Each source program was run on 1, 2,…, 7 CPUs 10 times each.

To get the accurate timing results *init* process whose pid is 1, was bound to CPU 0 using *setaffinity()*. Remaining CPUs were exclusively used for parallel lexical analysis for previous algorithm of parallel lexical analysis. For parallel Lexical analysis using memory blocks, OpenMP is used and same programs were considered for parallel lexical analysis. Maximum 7 threads were used for parallel lexical analysis.

Experimental results for various numbers of CPUs and various source codes have been collected but due to lack of space only tables for 10, 100,1000,10000 constructs have been given in Tables 1 through 4. The combined

effect on speedup is also shown in Tables 1 through 4 and Figures 6 through 9. In Tables 1-4 the meanings of columns various columns are as follows:

T_{Seq} : Time taken in sequential lexical analysis (in millisecond).
T_{PRR} : Time Taken in parallel lexical analysis using round robin scheduling (in millisecond).
T_{MB} : Time taken in parallel lexical analysis using memory blocks and OpenMP (in millisecond).
SP_{PRR}: Speed-up using round robin scheduling method.
SP_{MB} : Speed-up using memory blocks and OpenMP.
The speedup is calculated by using equation (6). For example, in Table 1, for 2 CPUs, the speedup for SP_{MB} is given as below:

$$\text{Speedup} = \frac{\begin{array}{c}\text{Time taken in}\\ \text{squential lexical anaysis}\\ \text{Time taken in parallel}\\ \text{lexical analysis}\\ \text{by 2 CPUs in parallel}\end{array}}{\begin{array}{c}\text{Time taken in squential}\\ \text{syntax lexical by 1 CPU}\\ \text{Time taken in parallel}\\ \text{lexical analysis}\\ \text{by 2 CPUs in parallel}\end{array}} = \frac{0.87}{0.68\ \text{Sec}} = 1.28 \quad (5)$$

Similarly speedup was computed for more number of processor.

Table 1. Time Taken and Speedup in Parallel Lexical Analysis of C Programs with 10 Construct in 2-7 CPUs/Threads

No of CPUs/No. of Threads	T_{Seq}	T_{PRR}	T_{MB}	SP_{PRR}	SP_{MB}
2	0.87	0.82	0.68	1.05	1.28
3	0.87	0.64	0.40	1.36	2.15
4	0.87	0.50	0.36	1.73	2.41
5	0.87	0.38	0.34	2.28	2.55
6	0.87	0.35	0.26	2.44	3.34
7	0.87	0.36	0.24	2.42	3.62

Table 2. Time Taken and Speedup in Parallel Lexical Analysis of C Programs with 100 Construct in 2-7 CPUs/Threads

No of CPUs/No. of Threads	T_{Seq}	T_{PRR}	T_{MB}	SP_{PRR}	SP_{MB}
2	6.66	5.96	3.64	1.11	1.82
3	6.66	4.04	2.65	1.64	2.51
4	6.66	3.08	2.06	2.15	3.22
5	6.66	2.52	1.96	2.64	3.38
6	6.66	2.04	1.67	3.26	3.97
7	6.66	1.85	1.64	3.58	4.05

Table 3. Time Taken and Speedup in Parallel Lexical Analysis of C Programs with 1000 Construct in 2-7 CPUs/Threads

No of CPUs/No. of Threads	T_{Seq}	T_{PRR}	T_{MB}	SP_{PRR}	SP_{MB}
2	79.44	69.12	22.87	1.14	3.47
3	79.44	45.86	18.08	1.73	4.39
4	79.44	34.63	15.20	2.29	5.22
5	79.44	27.78	13.97	2.85	5.68
6	79.44	23.21	13.06	3.42	6.08
7	79.44	19.85	12.19	4.00	6.51

Table 4. Time Taken and Speedup in Parallel Lexical Analysis of C Programs with 10000 Construct in 2-7 CPUs/Threads

No of CPUs/No. of Threads	T_{Seq}	T_{PRR}	T_{MB}	SP_{PRR}	SP_{MB}
2	1415.48	1386.28	456.95	1.02	3.09
3	1415.48	1062.65	312.41	1.33	4.53

4	1415.48	905.10	295.66	1.56	4.78
5	1415.48	808.31	229.59	1.75	6.16
6	1415.48	743.20	220.29	1.90	6.42
7	1415.48	699.22	206.87	2.02	6.84

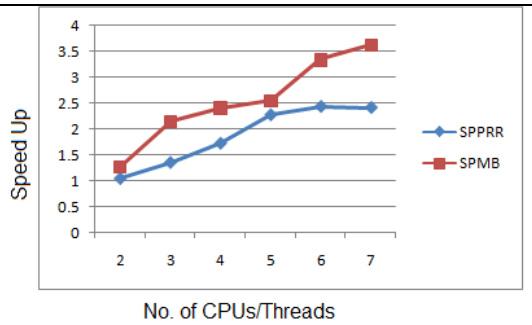

Fig. 6. Speedup in Parallel Lexical Analysis of C Programs with 10 Construct in 2-7 CPUs and Threads.

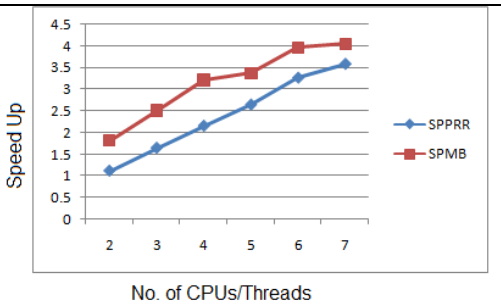

Fig. 7. Speedup in Parallel Lexical Analysis of C Programs with 100 Construct in 2-7 CPUs and Threads.

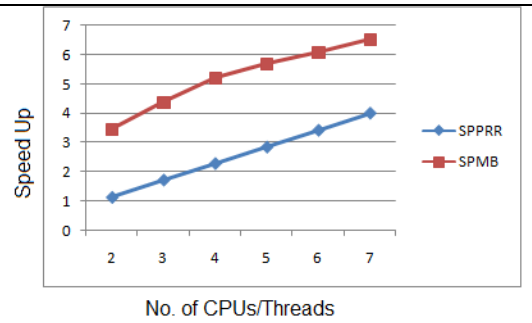

Fig. 8. Speedup in Parallel Lexical Analysis of C Programs with 1000 Construct in 2-7 CPUs and Threads.

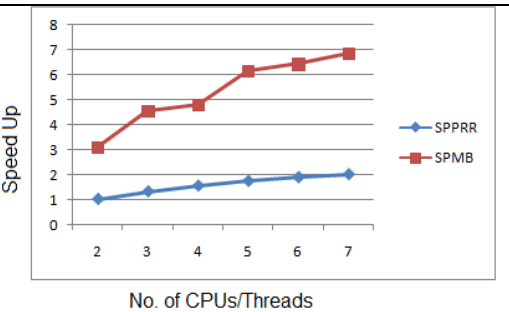

Fig. 9. Speedup in Parallel Lexical Analysis of C Programs with 10000 Construct in 2-7 CPUs and Threads.

8. Conclusion

In this paper we presented an improved version of parallel lexical analysis algorithm. It is clear from the results that the memory block algorithm outperforms the previously explored round robin approach of parallel lexical analysis. The maximum speedup achieved was 6.84. This speedup is expected to increase further if number of CPUs increases. As a consequence this approach would further improve the overall compilation time.

Appendix A. Source Code For Parallel Lexical Analysis Using OpenMP

```
%{
        char SourceFileName[100];
        int startbyte=0;int endbyte=0;
        int toktype;
%}
NL      [\n]
ID      [0-9]
DESH    [-]
Brackets        [{}()]
SPCILOP [!@#$%^&*-+/;"',._<>]
```

```
Void setup_buffers (const char *filename)
{
  unsigned char *file;
  size_t n_entries = 8, i = 0;
  int start_byte, end_byte;
  FILE *lineno_fp;
  file = mapf (filename);
  lineno_fp = fopen ("linenonew.txt", "r");
  buffers = malloc (n_entries * sizeof
(*buffers));
```

```
ARITHMETIC     [a-zA-Z]
SPACE   [ ]
%%
%%
char **buffers;
size_t n_buffers;
int main(int argc, char *argv[])
{
      setup_buffers(argv[1]);
      #pragma omp parallel for private(time)
       for (i = 0; i < n_buffers; i++)
        {
            yy_scan_string (buffers[i]);
           yylex();
        }
return 0;
}
size_t getfilesize (int fd)
{
  struct stat stbuf;
  fstat (fd, &stbuf);
  return (size_t) stbuf.st_size;
}
void *mapf (const char *filename)
{
  int fd;
  void *buf;
  size_t size;

  fd = open (filename, O_RDONLY);
  size = getfilesize (fd);
  buf = mmap (NULL, size, PROT_READ,
MAP_PRIVATE, fd, 0);
  return buf;
}
```

```
  while (fscanf (lineno_fp, "%d%d",
&start_byte, &end_byte) == 2)
  {
    if (i == n_entries)
    {
       n_entries *= 2;
       buffers = realloc (buffers, n_entries *
sizeof (*buffers));
    }
    buffers[i] = malloc (end_byte - start_byte
+ 2);
    strncpy (buffers[i], file + start_byte,
end_byte - start_byte + 1);
     i++;
  }
  n_buffers = i;
}
```

References

1. Marry Hall, David Padua, Keshav Pingali; "*Compiler Research: The Next 50 years*"; Communication of the ACM, Vol. 52, No. 2, pp. 60-67, February 2009.
2. David Gries; "*Compiler Construction for digital Computers*"; John Wiley & Sons Inc. USA, 1971.
3. Jean Paul Tremblay, Paul G. Sorenson; "*The Theory and Practice of Compiler Writing*"; McGraw-Hill Book Company USA, 1985.
4. Allen I. Holub; "*Compiler Design in C*"; Prentice Hall of India Pvt. Ltd., 1993.
5. Alfred V. Aho, Ravi Sethi, Jeffrey D.Ullman; "*Compilers: Principles, Techniques and Tools"*; Addison Wesley Publication Company, USA, 1986.
6. Alfred V. Aho, Ravi Sethi, Jeffrey D.Ullman; "*Principles of Compiler Design*"; Addison Wesley Publication Company, USA, 1985.
7. M. E. Lesk, E. Schmidt; "*Lex- A Lexical Analyzer Generator*"; Computing Science Technical Report No. 39, Bell Laboratories, Murray Hills, New Jersey, 1975.
8. http://flex.sourceforge.net/
9. Daniele Paolo Scarpazza, Gregory F. Russell;" High Performance regular expression scanning on Cell B.E. Processor; ICS 2009; pp. 14-25, 2009.
10. G. Umarani Shrikant ;"Parallel Lexical Analyzer on the Cell Processor"; IEEE SSIRI-C 2010; pp. 28-29;2010.
11. M. D. Mickunas, R. M. Schell; "*Parallel Compilation in a Multiprocessor Environment*"; Proceedings of the annual conference of the ACM, Washington, D.C., USA, pp. 241–246, 1978.
12. Barve, A.; Joshi, B.K.; , "A parallel lexical analyzer for multi-core machines," *(CONSEG), 2012 CSI Sixth International Conference on Software Engineering* ,pp.1-3, 5-7 Sept. 2012.

13. Barve, Amit; Joshi, Brijendra Kumar, "Parallel lexical analysis on multi-core machines using divide and conquer," *(NUiCONE), 2012 Nirma University International Conference onEngineering*, pp.1-5, 6-8 Dec. 2012.
14. http://openmp.org
15. Michael J. Quinn;"*Paralle Programming in C with MPI and OpenMP*";pp.159-160.Tata McGraw-Hill Publication, New Delhi 2003.
16. http://www.linuxjournal.com/article/6799?page=0,1.
17. http://www.cyberciti.biz/tips/setting-processor-affinity-certain-task-or-process.html
18. Barve, A.; Joshi, B.K.;"*Automatic C Code Generation for Parallel Compilation*"; International Journal on Advanced Computer Theory and Engineering (IJACTE); pp.26-28, **V**ol. 2, Issue 4, 2013